# Better abstraction for efficient code in HPC programs

YungYu Zhuang[1,a)] Shigeru Chiba[1,b)]

***Abstract:*** Slight code difference in HPC programs often results in significant performance change due to a variety of hardware and dedicated performance tuning techniques. If better abstraction for efficient code could be given, it is easier to revise or apply individuals of them for tuning performance on different platforms. However, HPC programs hardly take code abstraction into account since existing abstraction techniques tend to cause an impact on performance; efficient code might not be efficient anymore if it is implemented by existing abstraction techniques. Even though mixing the performance concern with others has drawbacks, separating the performance concern is meaningless unless no impact is introduced. We give examples in C language and incrementally improve abstraction to observe the impact caused by existing techniques. We also show our observation on performance change due to small code modification in different compilers and discuss future directions.

## 1. Introduction

This paper is a report on our research progress but not a research result. We point out the problem and show our experiment data, but have no solution yet. Toward the goal we discuss future directions.

Since the scale of problems to resolve and the programs targeted at them is going larger and larger, code abstraction is getting important. As to the evolution of code abstraction has a long history. It plays an important role in several topics related to modularity such as structured programming[2] and separation of concerns[3]. Without code abstraction it is difficult to model a big problem and modularize its implementation. A lot of language features, for example object-oriented programming and aspect-oriented programming, are thus developed for code abstraction and code modularity. Nevertheless, HPC programs are far from the evolution of programming languages since the language features used in HPC programs must be pure and simple enough to be optimized by the dedicated compiler for a specific platform. It might be one of the reasons why Fortran and C are preferred to implement HPC programs. Even though traditional programming languages such as Fortran are still growing and being extended, their users might choose to stay at an older version.

However, HPC programs are also facing the challenge of tackling complicated problems and optimizing the performance on various platforms. If a kind of code abstraction could be given, the programming conventions assumed by various compilers can be encapsulated and the techniques for tuning performance might be easier to compose or decompose, plug or unplug, reuse and share. It does not mean that hiding the tuning knowledge from HPC programmers but means that helping HPC programmers to tune performance with less efforts. HPC programmers can be aware of tuning techniques while quickly apply them. So far as we know, currently doing code abstraction in HPC programs relies on functions, macros, and libraries. We think it might be interesting to see whether such existing abstraction techniques can separate the performance concern or not, and thus conducted a series of small experiments on code abstraction in C language. In this paper we discuss neither C++ nor Java.

In the world of HPC the performance is the only metric to evaluate a program, but we were wondering if it is possible to improve code abstraction while let the performance as good as the one without code abstraction. We also have several observations on the performance change due to small code modifications. This motivates us to do several experiments on trying to abstract the code in order to avoid bugs and reuse efficient code.

## 2. Motivation

The code abstraction has several benefits such as better readability, better reusability, better maintainability, and better composability. It has been discussed in other domains for a long period and plays an important role in the evolution of programming language. However, there is room for improvement on code abstraction of HPC programs since in most cases code abstraction is not taken into account due to the concern about performance. Nevertheless, we assume that code abstraction is helpful to reuse efficient code, tune the performance, and avoid error-prone code. Here we focus on the readability and the reusability to explain what HPC programs can benefit from code abstraction.

### 2.1 Better readability

Since one of the purposes of HPC programs is simulation, vectors are often used. To provide better readability, the representation of the vector and its operations should be close to how they are written in the calculation on paper. However, they are usually described by an array of numbers or several arrays of numbers rather than an array of vector. The intention behide such repre-

[1] Dept. of Creative Informatics, The University of Tokyo
[a)] yungyu@acm.org
[b)] chiba@acm.org

```
1 static float pos[N*3];
```

(a)

```
1 static float pos_x[N];
2 static float pos_y[N];
3 static float pos_z[N];
```

(b)

**Fig. 1** Describing vectors by arrays of numbers

sentations might be the concern about performance, but here we would like to discuss how it should be if such code abstraction does not cause an impact on performance. For example, in a C program for N-body simulation we might define an array for storing the positions of all bodies as shown in Fig. 1(a), where N is the total number of bodies. Here all the positions of bodies are stored in the same array and each body occupies three floating-points to store its 3D coordinates. The vector for i-th body, (x, y, z), is got by (pos[i*3], pos[i*3+1], pos[i*3+2]). They are also usually stored by three separate arrays as shown in Fig. 1(b). In this case, the vector for i-th body is described by (pos_x[i], pos_y[i], pos_z[i]).

From the viewpoint of readability, such representations of vectors have several drawbacks. First, an operation on a vector has to be repeated three times to apply on each element of the vector. For example, to calculate the addition of two vectors, we need to add each element individually:

```
float x = pos_x[i] + pos_x[j];
float y = pos_y[i] + pos_y[j];
float z = pos_z[i] + pos_z[j];
```

Although the three statements should be put together to perform the addition of the two vectors, there is no such constraint on code-level. If the statement for z-direction is omitted or another statement is inserted between the statements for y-direction and z-direction, it might not be easy to know whether it is programmer's intention or a bug. Second, it is hard to separate them into a macro or a function since the number of values that have to be returned is three. A macro cannot return a value (without the extension supported by the GNU Compiler Collection[5]), and a function can only return a value or a pointer. In C language a widely used technique to overcome the limitation on the number of return value is passing the address of the variables to be written by parameters: however, variables must be prepared in advance on the call site. To improve the readability, we can use the structure supported in C language to represent a vector:

```
typedef struct {
  float x, y, z;
} Vec;
```

and encapsulate the operations on vectors into functions as shown in Fig. 2. Such a Vec can describe the vector as what it looks like on paper. The operations on vectors such as the addition are atomic and easy to understand. If a 2D version is needed, it is also easier—just modify the definition of Vec and the body of vec_add; no modification on the call site of vec_add is necessary.

### 2.2 Making efficient code reusable

Tuning the performance for HPC programs is never easy. Efficient code should be reusable since a small code modification often causes huge performance change. The reason is that a program can be optimized to get good performance only if its code

```
1 Vec vec_add(Vec first, Vec second) {
2   Vec ret;
3   ret.x = first.x + second.x;
4   ret.y = first.y + second.y;
5   ret.z = first.z + second.z;
6   return ret;
7 }
```

**Fig. 2** Separating the addition of vectors into a function

pattern can be accepted by the compiler. Programmers need to try different code patterns with the constraint that the logic for the other concerns cannot be changed. In order to get good performance, HPC programmers are forced to implement their code under a specific programming convention used inside the compiler for the platform to run on. Here we show our observations to explain.

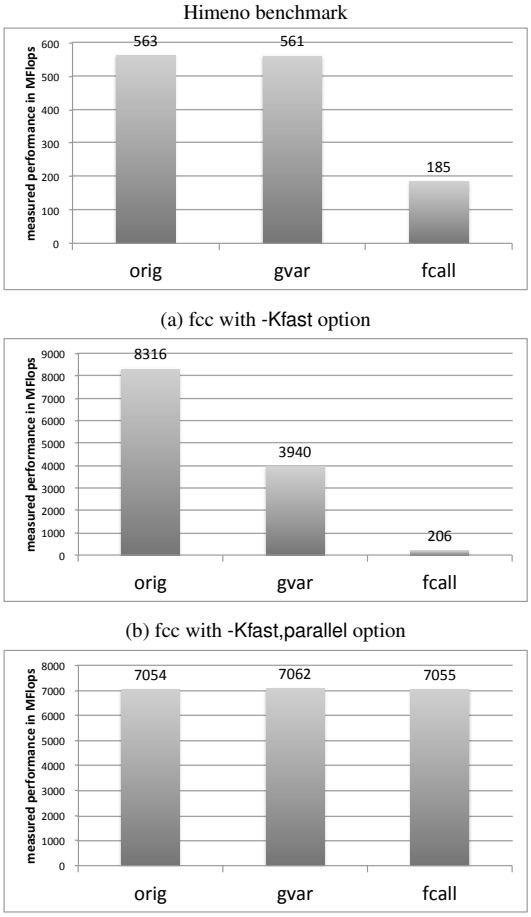
The first one was observed on the Fujitsu FX10 supercomputer[4]. We found that using a global variable for the reduction causes an impact on the optimization performed by the Fujitsu C compiler (fcc). However, such an impact was not observed when using the Intel C compiler (icc) on our machine. Here we use the Himeno benchmark[6] as an example. The one we used is the C static allocate version available on its web site. In the benchmark the function jacobi uses a local variable gosa to store the margin of error in the calculation, and returns the value of gosa when it ends. We modify this function to use a static global variable instead of a local variable, and let it return void:

```
static float gosa;
void jacobi(int nn) { ... }
```

We compare the performance of the global variable version with the original one, and the result is shown in Fig. 3(a) (named gvar and orig). They are compiled by fcc with -Kfast and run on one node of FX10 (SPARC64 IXfx 1.848 GHz processor with 16 cores)[10]. The result of comparison on parallel versions that are compiled with -Kfast,parallel is shown in Fig 3(b). Note that all the numbers used in this paper are the average of running the program ten times. Without the parallelization the measured performance before and after the modification are very close (orig: 563 MFlops, gvar: 561 MFlops). However, after the parallelization is applied, the results are quite different (orig: 8316 MFlops, gvar: 3940 MFlops). It shows that although the modification causes almost no impact when only -Kfast is applied, it prevents the program from being optimized for parallelization. Fig. 3(c) shows the result of compiling the same program by icc with -fast and running them on a machine with dual Intel Xeon E5-2687W processors (Sandy Bridge EP, 3.1GHz, 8 cores). In this case the impact is very small (orig: 7054 MFlops, gvar: 7062 MFlops).

Another performance impact due to small modification is also observed on FX10. Calling a function in the condition of loop causes an impact on the optimization performed by fcc. Again, we modify the Himeno benchmark as an example. In the function jacobi of the Himeno benchmark the loop for iterating all elements in the 3D grid looks like:

```
for(i=1 ; i<imax-1 ; i++)
  for(j=1 ; j<jmax-1 ; j++)
    for(k=1 ; k<kmax-1 ; k++){
      :
    }
```




(a) fcc with -Kfast option

(b) fcc with -Kfast,parallel option

(c) icc with -fast option

**Fig. 3** The difference on performance caused by using a global variable or a function call in loop condition

```
int getimax() { return imax; }
int getjmax() { return jmax; }
int getkmax() { return kmax; }
```

**Fig. 4** The accessor functions for imax, jmax, and kmax

If the variable accesses imax, jmax, and kmax are replaced with their accessors, i.e. getimax(), getjmax(), and getkmax() as shown in Fig. 4, the loop cannot be optimized by fcc well. As shown in Fig. 3(a), the measured performance after this modification (named fcall) is about 185 MFlops. The measured performance of the parallelization version is about 206 MFlops as shown in Fig. 3(b). Although we expect function inlining will be performed by the compiler and then cause no impact on performance, the result shows that there is an impact on performance due to this modification. After checking the optimization log generated by fcc, we found that indeed function inlining is done but it is performed after the loop optimization. Since the function calls has not been inlined yet when the loop optimization is performed, the condition looks too complicated to optimize. This unexpected performance impact is hardly classified as a bug of fcc and can be regarded as a result of violating the programming convention used inside fcc. Since the code does not match the pattern assumed by fcc, the optimization for this code cannot be applied. On the other hand, Fig. 3(c) shows that there is almost no difference on performance between orig version and fcall version.

Similar things can be observed on icc as well. If we write a

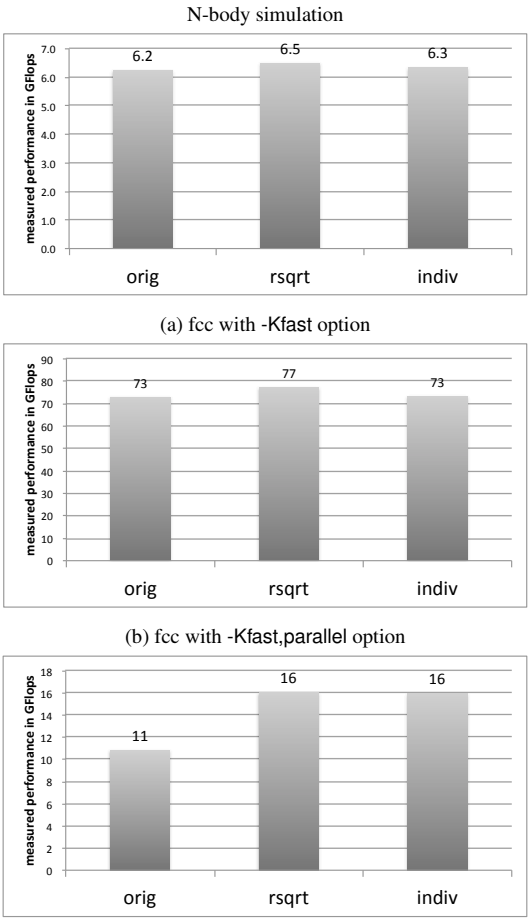
```
float a = sqrtf(rx * rx + ry * ry + rz * rz + 0.01f);
float s = pos1[i].w / (a * a * a);
```

(a)

```
float a = 1.0f / sqrtf(rx * rx + ry * ry + rz * rz + 0.01f);
float s = pos1[i].w * (a * a * a);
```

(b)

**Fig. 5** Multiplying by the result of dividing 1.0f by sqrt instead of dividing by sqrt




(a) fcc with -Kfast option

(b) fcc with -Kfast,parallel option

(c) icc with -fast option

**Fig. 6** The difference on performance caused by taking advantage from rsqrt and using individual float arrays

code that matches the code pattern inside icc, the performance can be greatly improved. Here we use the N-body simulation as an example again. In the program we have two lines of code that using sqrtf as shown in Fig. 5(a). We may modify it to take advantage of function returning the reciprocal of the square root (rsqrtf) provided by the compiler as shown in Fig. 5(b). In other words, multiplying by the result of dividing 1.0f by sqrt instead of dividing by sqrt. The performance of the two versions (orig and rsqrt) are shown in Fig. 6. When using icc to compile, the performance of rsqrt version compiled by icc is much faster than the original one as shown in Fig. 6(c) (16 GFlops vs. 11 GFlops). However, as shown in Fig. 6(a) and Fig. 6(b), such difference is not obvious when using fcc.

The last example we are going to show is observed on icc as well. As we mentioned in Sec. 2.1, to store the coordinates for objects in N-body simulation we may use an array of structure or individual arrays. The original version discussed in the previous

example adopts an array of structure to store the 3D coordinates and the weight. The indiv version shown in Fig. 6 shows the performance of using 4 individual arrays rather than an array of a structure that contain 4 variables. We can find that in the case of icc the performance is greatly improved (orig: 11 GFlops, indiv: 16 GFlops). However, such a significant difference is not observed on fcc.

These examples show that the performance of a program heavily depends on whether its code can match the code pattern that can be optimized by the compiler. In other words, we have to follow the specific programming convention assumed by the compiler that we are going to use. Since in the world of HPC different platforms usually need different dedicated compilers, we are forced to write different versions based on different programming conventions in order to get the best performance on those platforms. This motivates us to attempt to abstract the code for hiding the difference between various programming conventions. If such code abstraction could be given, it is easier to switch between them for performance tuning. It is not to say that programmers should be unaware of such programming convention, but means that programmers can improve the code for the performance concern without modifying other code and further reuse the tuning techniques in other programs.

## 3. Code abstraction with existing techniques

This section is a series of experiments on whether the performance concern can be separated well by existing abstraction techniques. In Sec. 2.2 the last example shows that the performance of the version with code abstraction using struct (orig version) seems worse than the one without code abstraction (indiv version), though the difference depends on the compiler. In order to know such an impact on performance due to code abstraction, we implement abstraction for readability and reusability in two programs with existing techniques. From the result of this experiments we know that the performance concern cannot be separated well by existing abstraction techniques. Below we explain our modification for abstraction and show the numbers.

### 3.1 The abstraction for readability

The abstraction for readability can be done, but it is very slow. The example we want to show is making the N-body simulation program we mentioned in Sec. 2 more readable by using functions, structures, and pointers. In the program we have two buffers as shown in Fig. 7 and have functions map1 and sum1 for array pos1, map2 and sum2 for array pos2. The purpose of the functions map1 and map2 is performing the motion for every body in one round, and the functions sum1 and sum2 are respectively called inside map1 and map2 to sum the forces on a body. The operations inside map1 and map2, sum1 and sum2 are totally the same except the arrays they operate. This is a little confusing since the code reviewer might not sure if the two functions do really the same thing. Furthermore, ensuring the consistency between such duplicated functions takes efforts. To make the intention clear we merge them into two functions map and sum, and use pointer to indicate which array to operate. Then we further improve code abstraction by encapsulating vector operations. In the original program, although we use struct to declare the buffers, the operations on vectors are performed by calculating each coordinate individually as shown in Fig. 8(a). As discussed in Sec. 2.1, vector operations can be encapsulated to improve code abstraction. We extract vector operations into functions, for example vec3_sub and vec3_mult, and rewrite them as shown in Fig. 8(b).

```
typedef struct {
  float x, y, z, w;
} Vec4;
static Vec4 pos1[N];
static Vec4 pos2[N];
```

**Fig. 7** The buffers used to store the 3D coordinates and the weight in our N-body simulation program

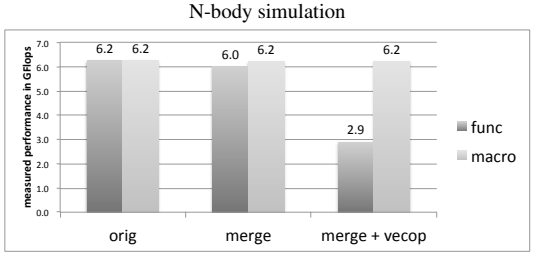
```
float rx = p.x - pos[j].x;
float ry = p.y - pos[j].y;
float rz = p.z - pos[j].z;
float a = sqrtf(rx * rx + ry * ry + rz * rz + 0.01f);
```

(a)

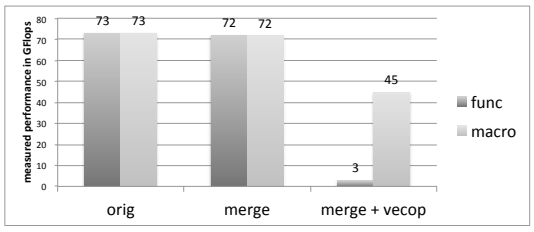
```
Vec3 r = vec3_sub(p, pos[j]);
float a = sqrtf(vec3_mult(r, r) + 0.01f);
```

(b)

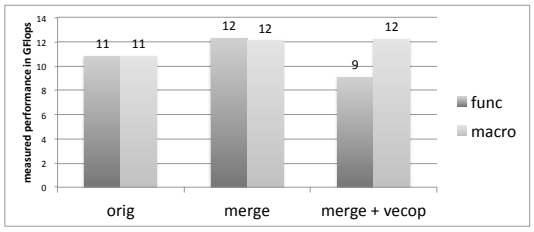
**Fig. 8** Encapsulating vector operations in functions




(a) fcc with -Kfast option




(b) fcc with -Kfast,parallel option




(c) icc with -fast option

**Fig. 9** The performance impact caused by merging functions and encapsulating vector operations in this example

The measured performance of the three versions: the original one (named orig), the one that merges functions (named merge), and the one that further encapsulates vector operations (named merge + vecop), is shown by the dark grey bars (marked with func) in Fig. 9. From the result we can know the two modifications we made lowered the performance, especially the one that encapsulates vector operations. In the case of fcc without paral-

lelization, the measured performance of orig is about 6.2 GFlops, the one of merge is about 6.0 GFlops, and the one of merge + vecop is about 2.9 GFlops. In the case of fcc with parallelization, the measured performance of them are about 73 GFlops, 72 GFlops, and 3 GFlops, respectively. As to the case of icc they are about 11 GFlops, 12 GFlops, and 9 GFlops, respectively. The reason might be due to the cost of calling functions and passing structures, and the result of preventing the optimization. Another interesting observation is that merging functions even slightly improves the performance when using icc.

Using macros might be helpful, but its overheads are not zero. The light grey bars marked with macro in Fig. 9 show the performance of using macros instead of functions to implement the above code abstraction. It shows that using macros can preserve the performance in both the cases of fcc and icc, but has an impact on fcc parallelization (for fcc without parallelization all the three versions are about 6.2 GFlops, for icc they are about 11 GFlops, 12 GFlops, and 12 GFlops, while for fcc with parallelization they are about 73 GFlops, 72 GFlops, and 45 GFlops). Although this experiment shows that the impact caused by macros is much smaller than the one caused by functions and structures, it is still not zero. Furthermore, without the GCC extension the macro in C does not allow returning a value as what the function does; its usage is limited. Type checking and debugging is also difficult. Using inline functions might be better but its overheads are not zero, either. To sum up, existing techniques can be used to abstract the code for readability, but the overheads caused by them cannot be totally eliminated.

### 3.2 The abstraction for reusability

In this subsection we want to abstract the code for reusability. As we discussed in Sec. 2.2, writing code abstraction for reusing efficient code is one of our motivations. However, with existing abstraction techniques the abstraction for reusability might be done, but the implementation is not fast ever. Here we use the Himeno benchmark again to demonstrate the abstraction for reusability and show its impact on performance. When we are tuning the performance of our remake of the Himeno benchmark based on our framework on FX10, we got a lot of advice from Fujitsu. One of them is swapping the indexes of the 4D array. As shown in Fig. 10(a) A 4D array in the Himeno benchmark actually contains several 3D arrays, and the indexes (n, i, j, k) are used to indicate the (i, j, k) element in n-th 3D array. The idea is to swap the indexes to (i, j, k, n). Since which one is faster depends on the access pattern and the memory available on the platform, we define an array and its accessor as shown in Fig. 10(b) to make it easy to swap the indexes for performance tuning on a specific platform. Now the client side of Float4Array always uses the accessor FLOAT4ARRAY to access an element, and thus changing the order of the indexes inside Float4Array does not cause additional modification on the client side; the tuning can be easily done by modifying Fig. 10(b) to Fig. 10(c). The next we are going to do is abstracting the kernel calculation. We were wondering if we can quickly write similar stencil programs by simply replacing the kernel calculation. If it is possible, we can reuse the code that we have tuned. A well-known existing technique

```
1 static float  a[4][MIMAX][MJMAX][MKMAX],
2               b[3][MIMAX][MJMAX][MKMAX],
3               c[3][MIMAX][MJMAX][MKMAX];
```

(a)

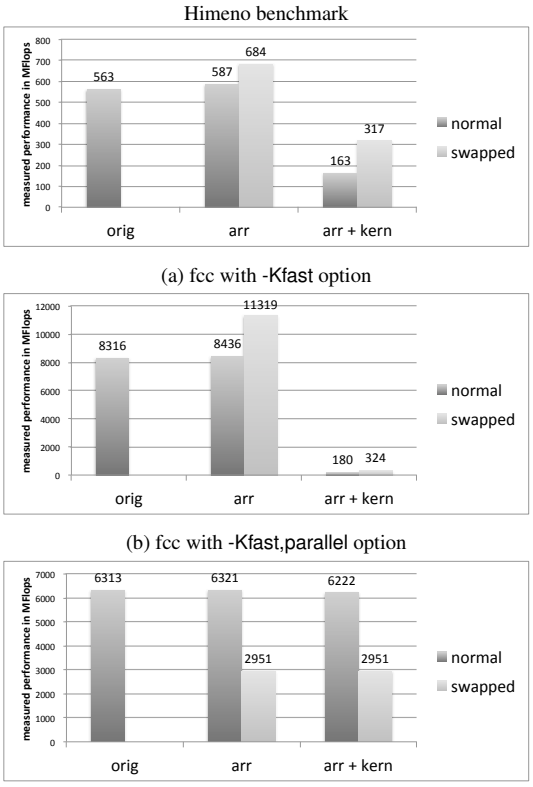
```
1 typedef float Float4Array[4][MIMAX][MJMAX][MKMAX];
2 #define FLOAT4ARRAY(a, n, i, j, k) (a[n][i][j][k])
3 static Float4Array  a, b, c;
```

(b)

```
1 typedef float Float4Array[MIMAX][MJMAX][MKMAX][4];
2 #define FLOAT4ARRAY(a, n, i, j, k) (a[i][j][k][n])
3 static Float4Array  a, b, c;
```

(b)

**Fig. 10** Defining an array and its accessor to make it easy to swap the indexes




(a) fcc with -Kfast option

(b) fcc with -Kfast,parallel option

(c) icc with -O3 -mcmodel=medium option

**Fig. 11** The performance impact caused by defining arrays and using function pointer for kernel calculation in this example

to do such thing is function pointer, while it is also known for its performance impact. To know how much it costs we further extract the kernel calculation by declaring the following function pointer:

```
static float
(*kernel)(int i, int j, int k, float* r);
```

and extract the kernel calculation to a function named my_kernel and assign the address of my_kernel to the function pointer kernel:

```
kernel = &my_kernel;
```

The place that executing the kernel calculation is replaced with the call to the function pointed by kernel:

```
(*kernel)(i, j, k, &gosa);
```

where i, j, k, and &gosa are the arguments given to the function.

Fig. 11 shows the result. The drak grey bars marked with normal refer to the versions without swapping the indexes, while

the light bars marked with swapped refer to the version with swapped indexes. If we look at the first step that defining arrays (named arr), it is successful. For the case of fcc without parallelization, the performance of the original one is about 563 MFlops, the modified version (named arr) is about 587 MFlops, and the one after swapped is about 684 MFlops. For the case of fcc with parallelization, the performance of them are about 8316 MFlops, 8436 MFlops, and 11319 MFlops, respectively. On the other hand, swapping the indexes decreases rather than increases the performance in the case of icc. Note that due to the memory limitation we have to specify the option -mcmodel=medium and use -O3 instead of -fast. It is not surprising that the performance of using function pointer (named arr + kern) is very low for the case of fcc either with or without parallelization. The numbers are about 163 MFlops and 180 MFlops, respectively. Even after the indexes are swapped, they are about 317 MFlops and 324 MFlops, respectively. In the case of icc, however, it does not cause a notable impact (arr: 6321 Mflops, arr + kern: 6222 MFlops. It might be related to memory model or optimization, and shows that such performance tuning needs analysis and try for a specific platform. Readers might be wondering if macros can be used to implement such code abstraction with limited performance impact as we showed in previous subsection. Unfortunately, so far as we know, the macro in C is not powerful enough to do such abstraction.

The experiment result points to an interesting but sad conclusion: with existing abstraction techniques, if we want to abstract the efficient code for reusability, the code will be no longer efficient. Given that you have an idea on performance improvement, that idea does not work anymore if you implement that idea by a library or something else. Indeed class library might be a possible solution to separate the performance concern, but its flexibility is quite limited if function pointer cannot be used. On the other hand, once we use function pointer, the abstraction for the performance concern becomes meaningless.

## 4. Related work and future directions

Although in Sec. 3 we got a discouraging conclusion that the efficient code cannot be abstracted, there is an assumption that using existing abstraction techniques such as typedef, struct, macros, functions, and function pointers. If we look into the reason why they cannot abstract the code well, we can find the root cause is that the abstracted code does not match the code pattern used inside the compilers. In other words, existing abstraction techniques allow programmers to abstract the code but cannot let the code be accepted by the compilers. If there were a way to write down the code with abstraction but translate them to the code that matches the code pattern before giving to the compilers, we could enjoy both code abstraction and efficiency.

### 4.1 Metaprogramming

In fact, there is such a technique—metaprogramming. It has been developed for a long time, but so far as we know it has not been used to abstract the code for HPC programs yet. Metaprogramming is powerful but complicated. Metaobject protocols[8], the template in C++[1], the macro in Lisp, and other metaprogramming techniques can be used to separate the performance concern by translating code while preserving the runtime performance. The dynamic parts that tend to cause an impact on performance can be statically translated at compile-time and thus their runtime overheads can be eliminated. However, how to make it easy to use is still a challenge. Programmers need to consider their programs at meta-level. Moreover, debugging is also a difficult part to tackle.

### 4.2 Domain-specific language

Domain-specific language (DSL)[7] can be regarded as a compromise since it simplifies the usage of metaprogramming and provides limited code translation support. Research activities such as Physis[9] can be classified into this category. With DSL it is possible to give a limited set of language features and translate the DSL code to the code that can be optimized by the dedicated compiler for a specific platform. The problem of DSL approach is the flexibility. When a DSL does not fit the needs, it takes efforts to modify the DSL specification and its compiler.

On the other hand, class libraries are easier to use and modify. If a DSL can be customized as easy as a class library, it might be able to separate the performance concern from other concerns while rapidly customizing the DSL based on the needs found during the progress of development. That is what we are developing, a DSL framework named Bytespresso. Bytespresso combines DSL with metaprogramming to allow programmers to develop their own DSLs and translate the DSL code to the one that matches the code pattern accepted by the compilers. How to avoid increasing its complexity while giving full flexibility is our challenge.

## 5. Conclusions

Since HPC programs are getting more and more complicated due to complex tuning techniques, even a small modification tends to cause either a bug or an impact on performance. We showed how a slight code difference may cause a significant performance change on different platforms by several examples. We suggested that such a code difference should be regarded as a performance concern and be separated from other concerns to make it easier to do performance tuning on various platforms without touching the code for other concerns. We also explained what we can benefit from code abstraction and then attempted to use existing techniques to abstract the code. However, unfortunately the experiments showed a discouraging result that code abstraction tends to cause a performance impact. With existing abstraction techniques, whenever we want to abstract efficient code, the code becomes no longer efficient. The reason is that the abstracted code is far from the programming convention that compilers can optimize. To tackle this issue, code translation is a promising solution since the abstracted code can be translated to the code pattern accepted by the compilers. Metaprogramming and domain-specific language are possible approaches, but how to reduce the complexity of usage is still a challenge.

## References

[1] Abrahams, D. and Gurtovoy, A.: *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*, Addison-Wesley Professional (2004).
[2] Dahl, O. J., Dijkstra, E. W. and Hoare, C. A. R.(eds.): *Structured Programming*, Academic Press Ltd., London, UK, UK (1972).
[3] Dijkstra, E. W.: The Structure of the &Ldquo;THE&Rdquo;-multiprogramming System, *Commun. ACM*, Vol. 11, No. 5, pp. 341–346 (online), DOI: 10.1145/363095.363143 (1968).
[4] Fujitsu: PRIMEHPC FX10, Fujitsu Limited (online), available from ⟨http://www.fujitsu.com/jp/products/computing/servers/supercomputer/primehpc-fx10/⟩ (accessed 2015-07-09).
[5] GNU Compiler Collection (GCC): Statements and Declarations in Expressions, GNU (online), available from ⟨https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html#Statement-Exprs⟩ (accessed 2015-07-09).
[6] Himeno, R.: Himeno benchmark, Source code（C, static allocate version）, Advanced Center for Computing and Communication, RIKEN (online), available from ⟨http://accc.riken.jp/2465.htm#itemid4535⟩ (accessed 2015-07-09).
[7] Hudak, P.: Modular domain specific languages and tools, *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pp. 134–142 (online), DOI: 10.1109/ICSR.1998.685738 (1998).
[8] Kiczales, G. and Rivieres, J. D.: *The Art of the Metaobject Protocol*, MIT Press, Cambridge, MA, USA (1991).
[9] Maruyama, N., Nomura, T., Sato, K. and Matsuoka, S.: Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-scale GPU-accelerated Supercomputers, *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, ACM, pp. 11:1–11:12 (online), DOI: 10.1145/2063384.2063398 (2011).
[10] Oakleaf-FX,Oakbridge-FX: FX10 Supercomputer system, Supercomputing Division, Information Technology Center, The University of Tokyo (online), available from ⟨http://www.cc.u-tokyo.ac.jp/system/fx10/⟩ (accessed 2015-07-09).